



University of  
Pittsburgh

# Algorithms and Data Structures 2

## CS 1501



Spring 2023

Sherif Khattab

[ksm73@pitt.edu](mailto:ksm73@pitt.edu)

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

# Announcements

- Upcoming Deadlines
  - Homework 7: this Friday @ 11:59 pm
  - Lab 5: Tuesday 2/28 @ 11:59 pm
  - Assignment 2: Friday 3/17 @ 11:59 pm
    - Support video and slides on Canvas
- Midterm Exam on Wednesday 3/1
  - in-person, closed-book
  - Study guide and old exams on Canvas
  - Practice questions on GradeScope (with answers)
- Midterm Review Session today at 5:30 pm
  - Recorded
  - Same Zoom link as Student Support Hours

# Previous lecture

- Huffman Compression

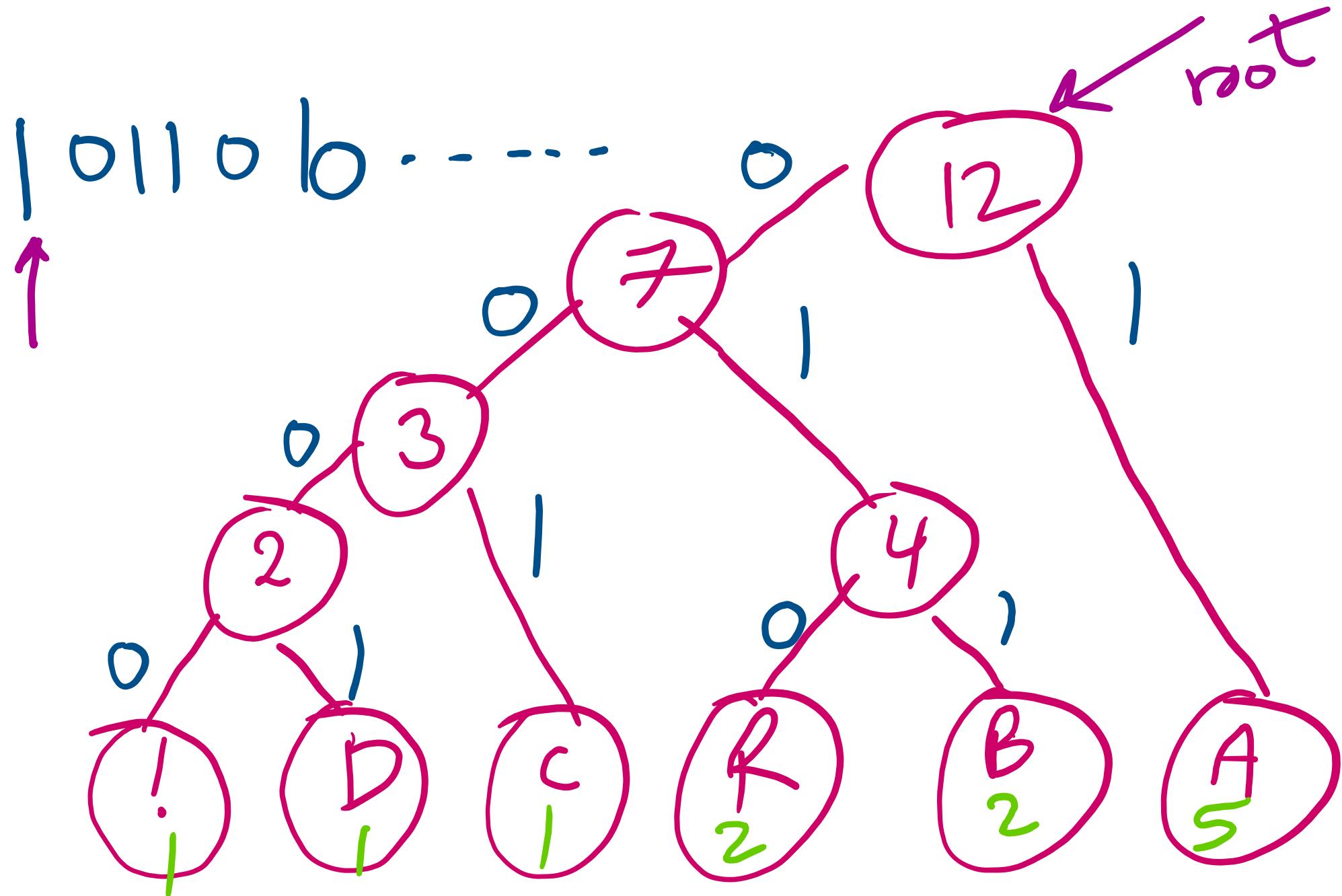
# This Lecture

- Huffman Compression
- Run-length Encoding
- LZW

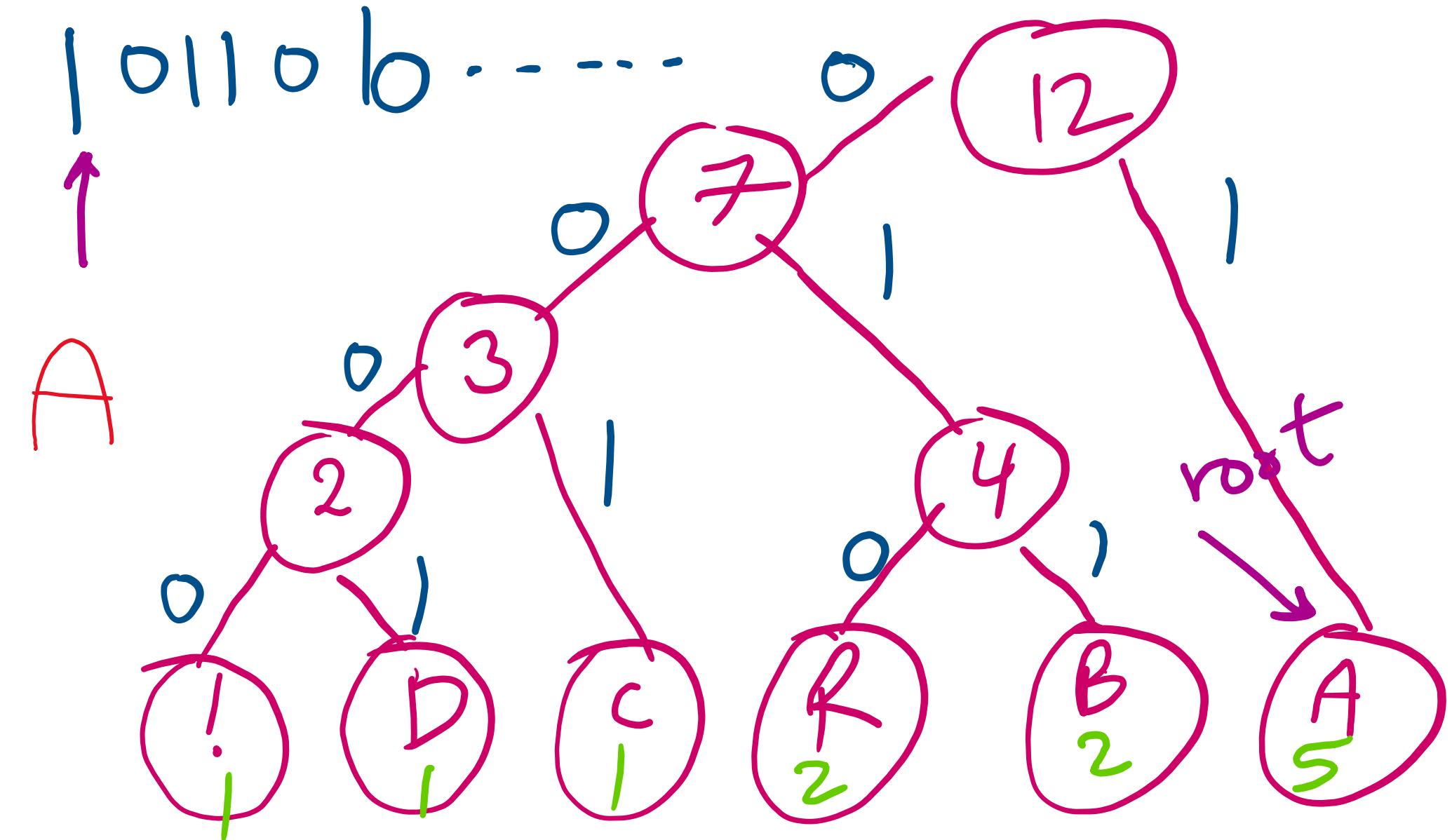
# Huffman pseudocode

- Encoding approach:
  - Read input
  - Compute frequencies
  - Build trie/codeword table
  - Write out trie as a bitstring to compressed file
  - Write out character count of input (**why is that necessary?**)
  - Use table to write out the codeword for each input character
- Decoding approach:
  - Read trie
  - Read character count
  - Use trie to decode bitstring of compressed file

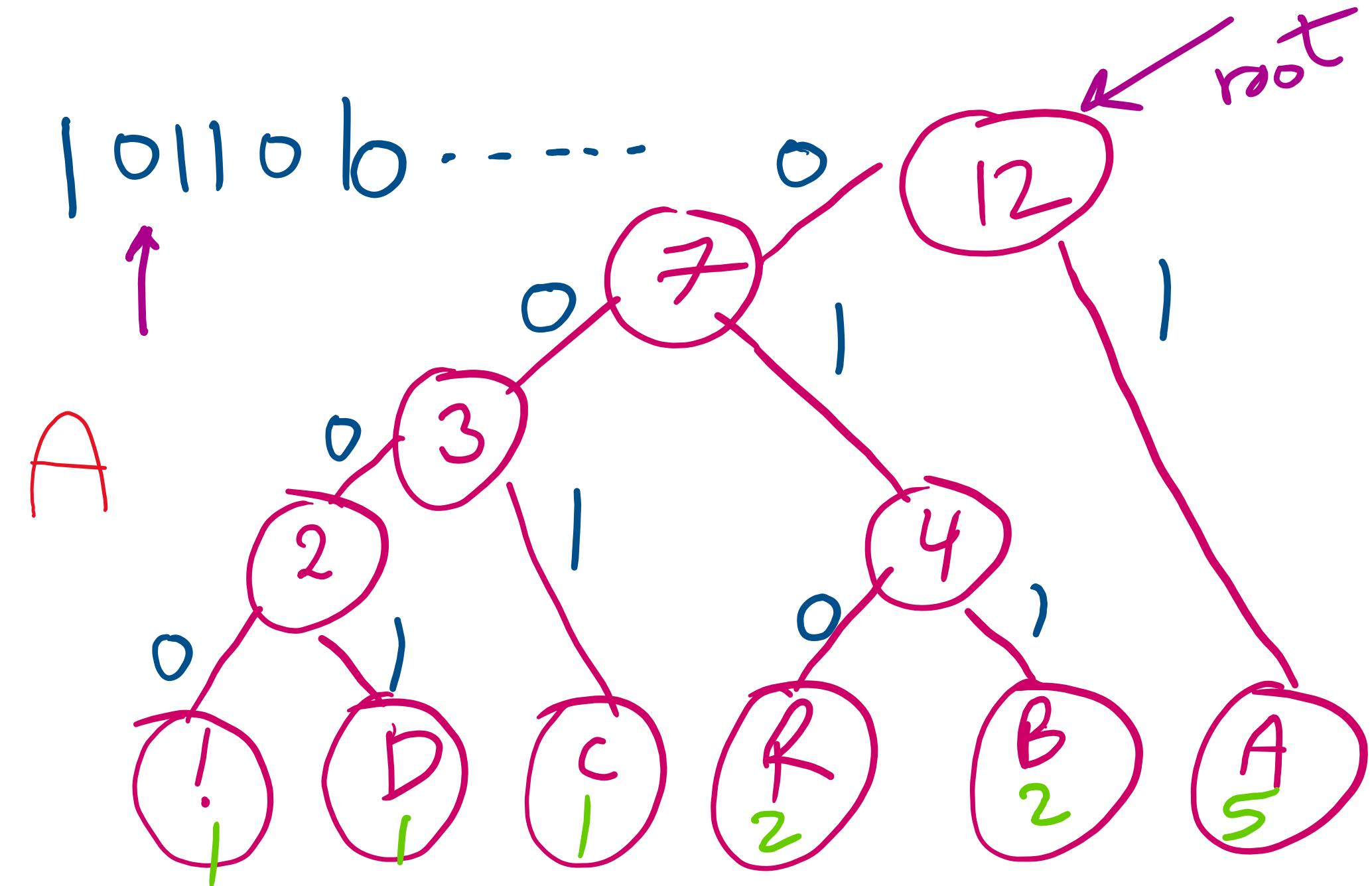
# Huffman Compression: Decoding



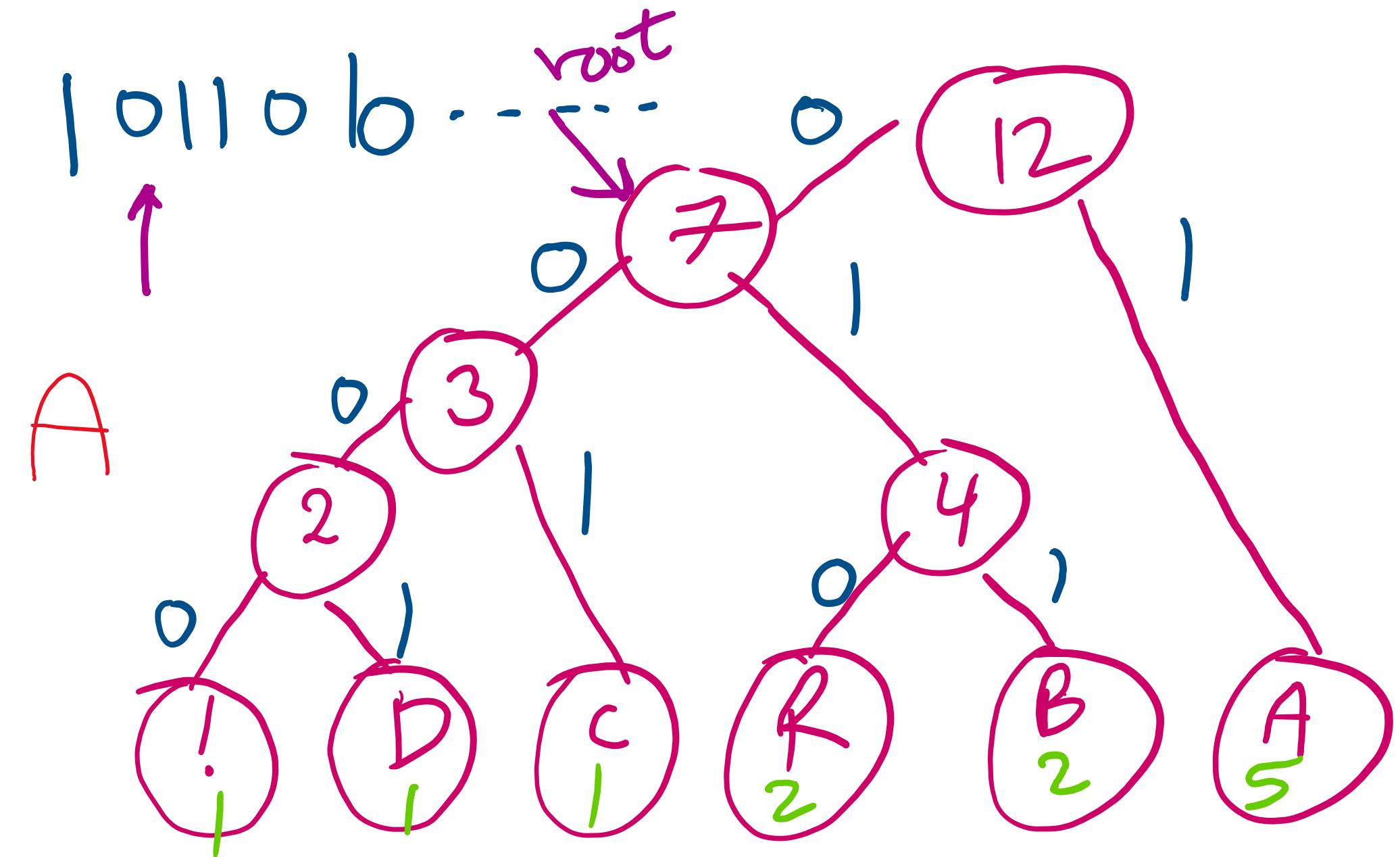
# Huffman Compression: Decoding



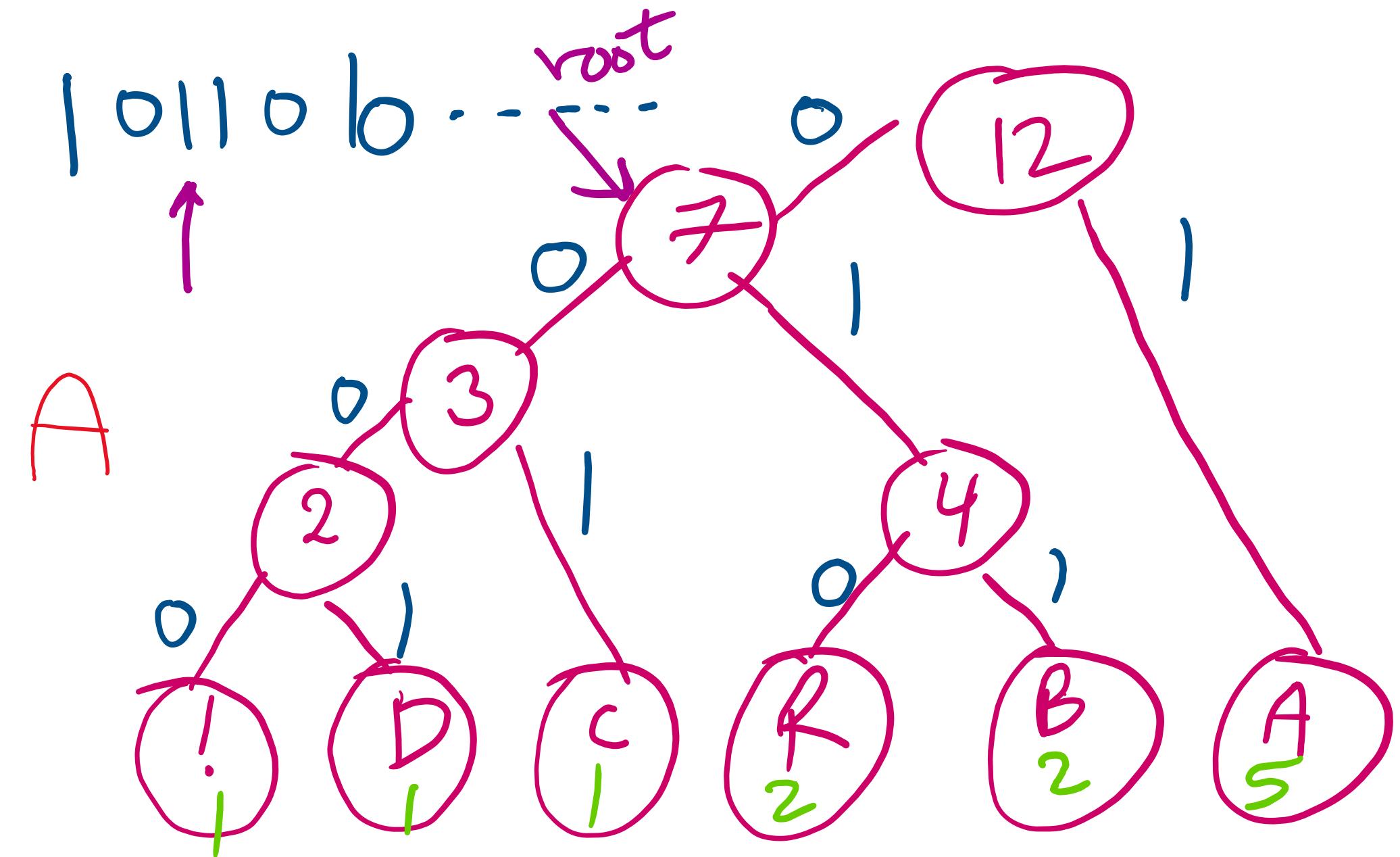
# Huffman Compression: Decoding



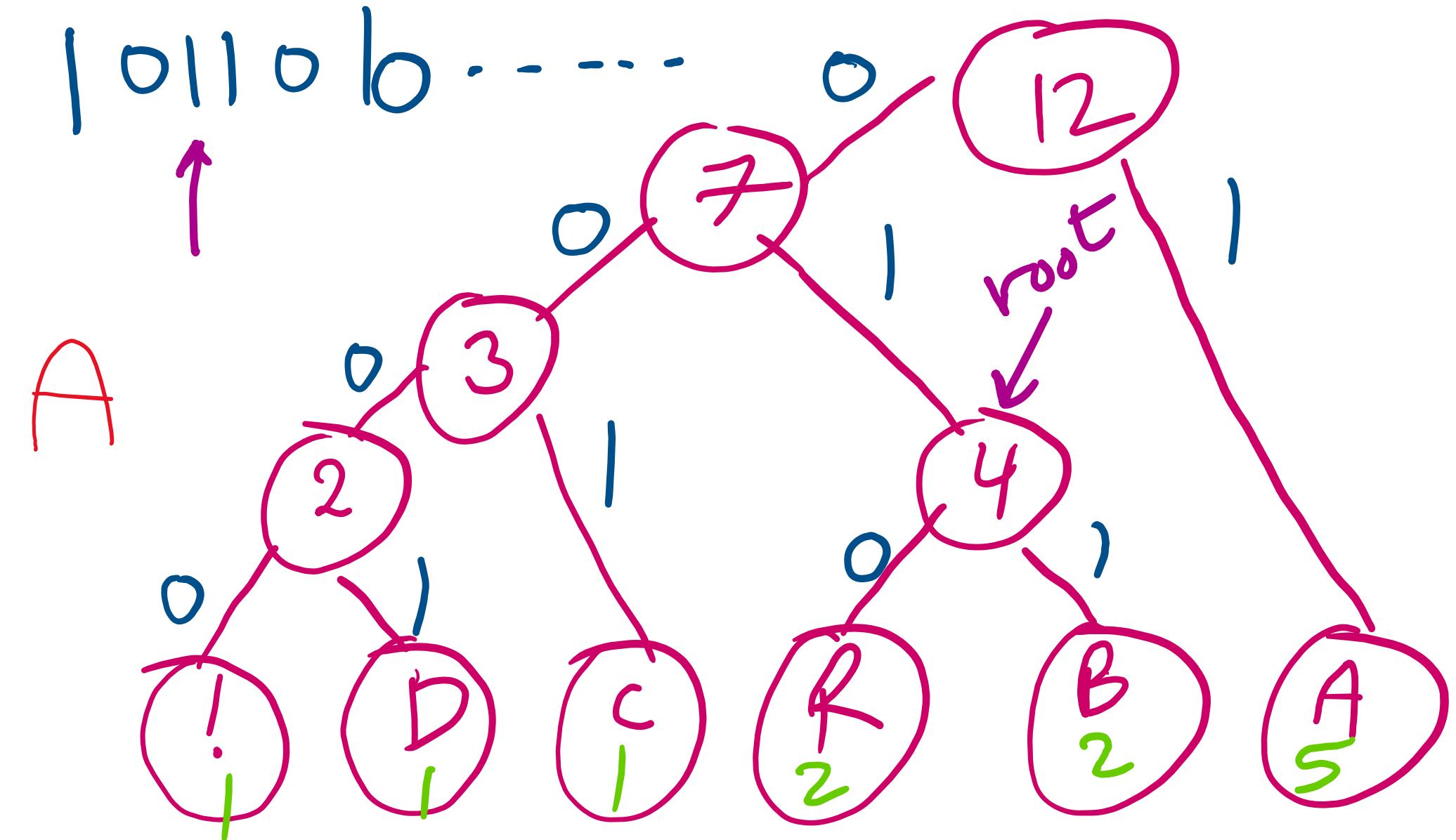
# Huffman Compression: Decoding



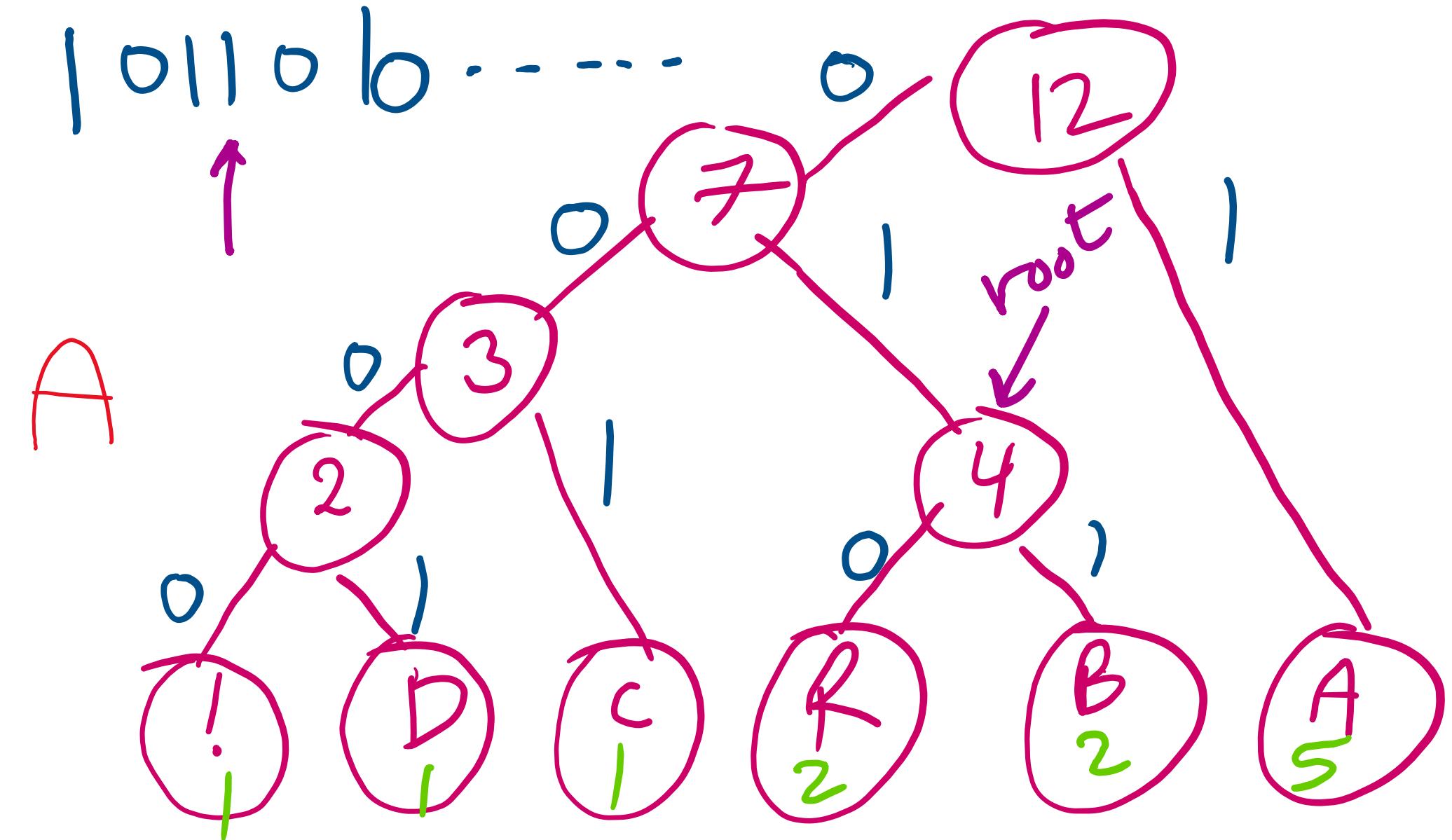
# Huffman Compression: Decoding



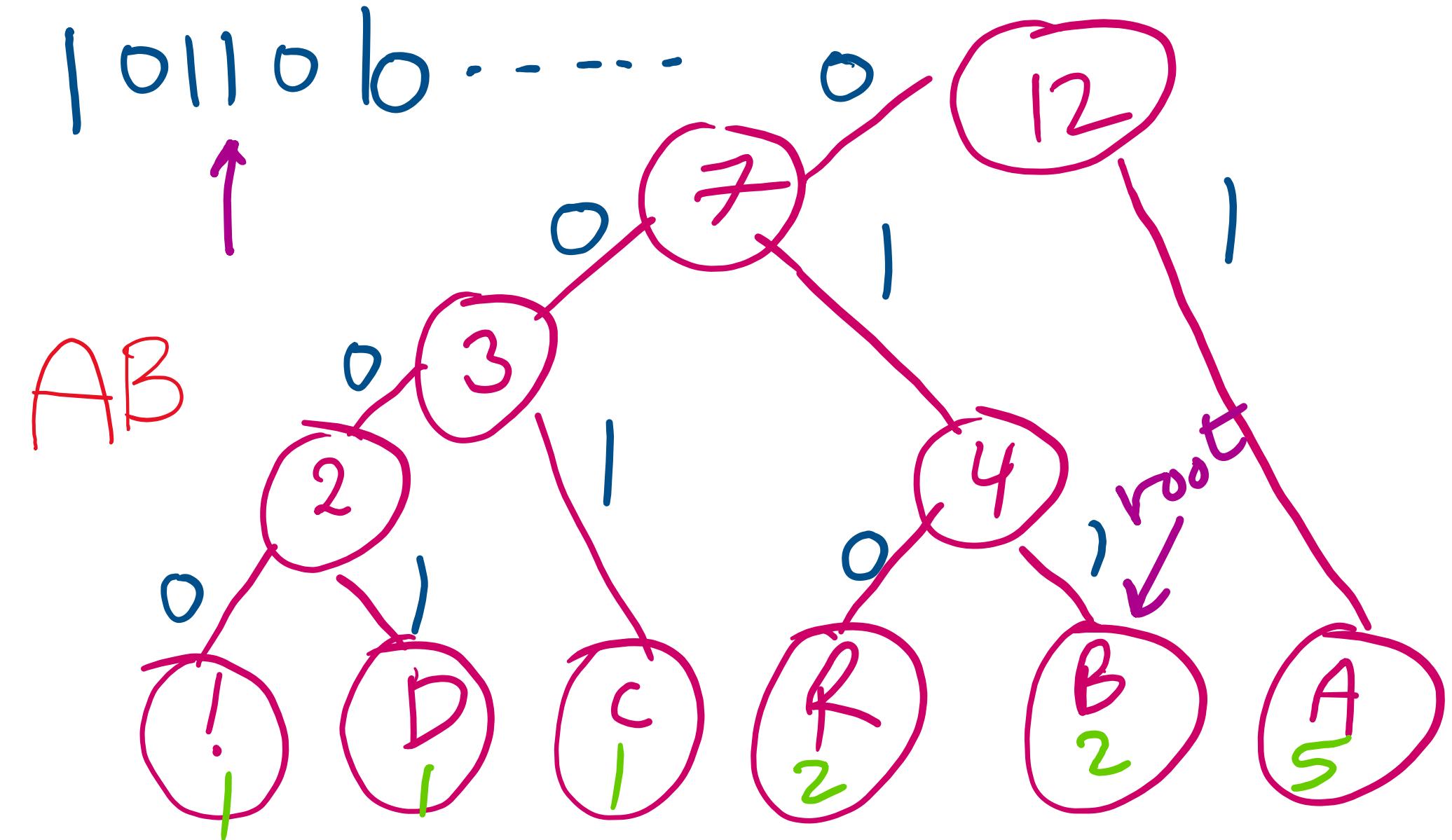
# Huffman Compression: Decoding



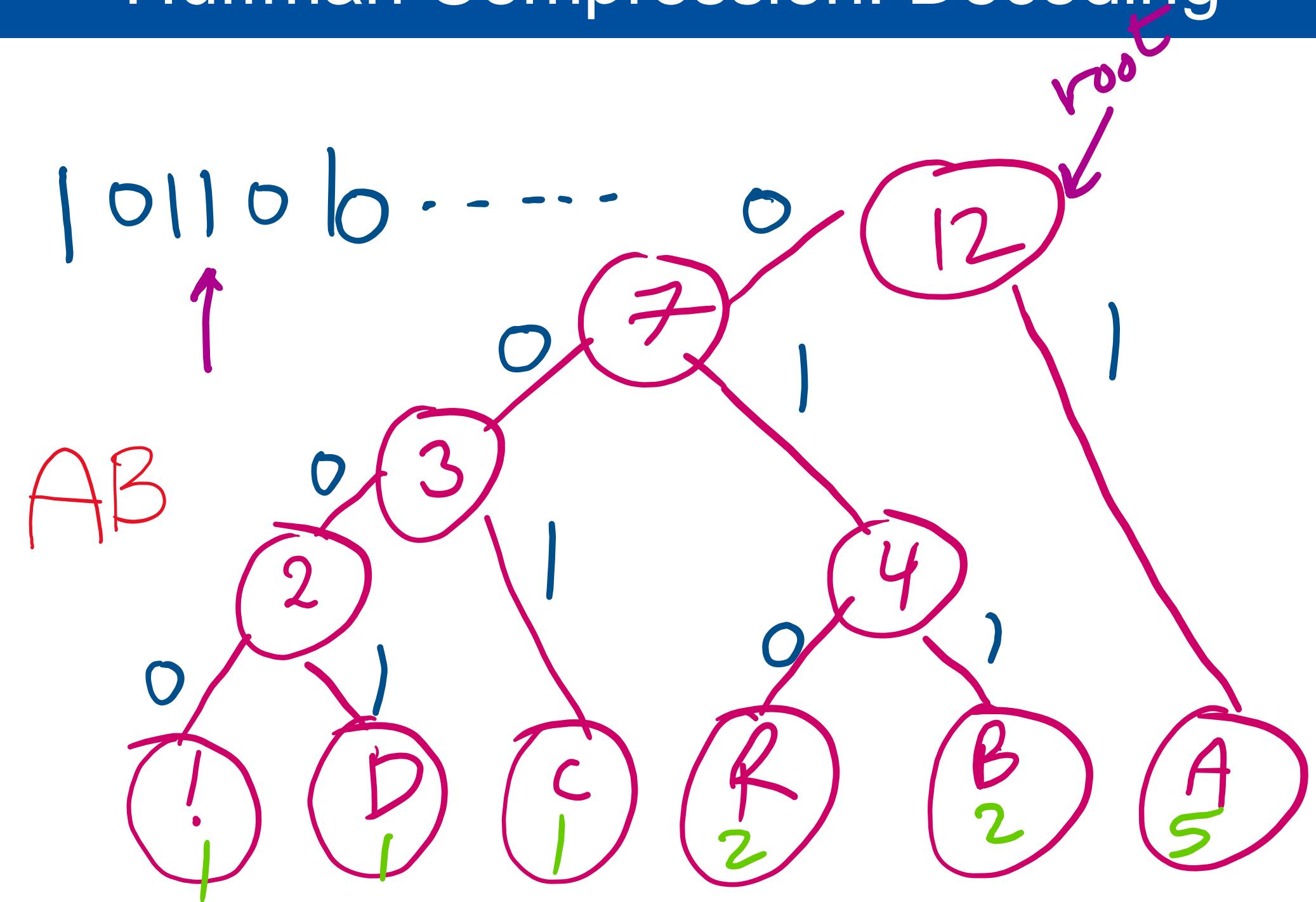
# Huffman Compression: Decoding



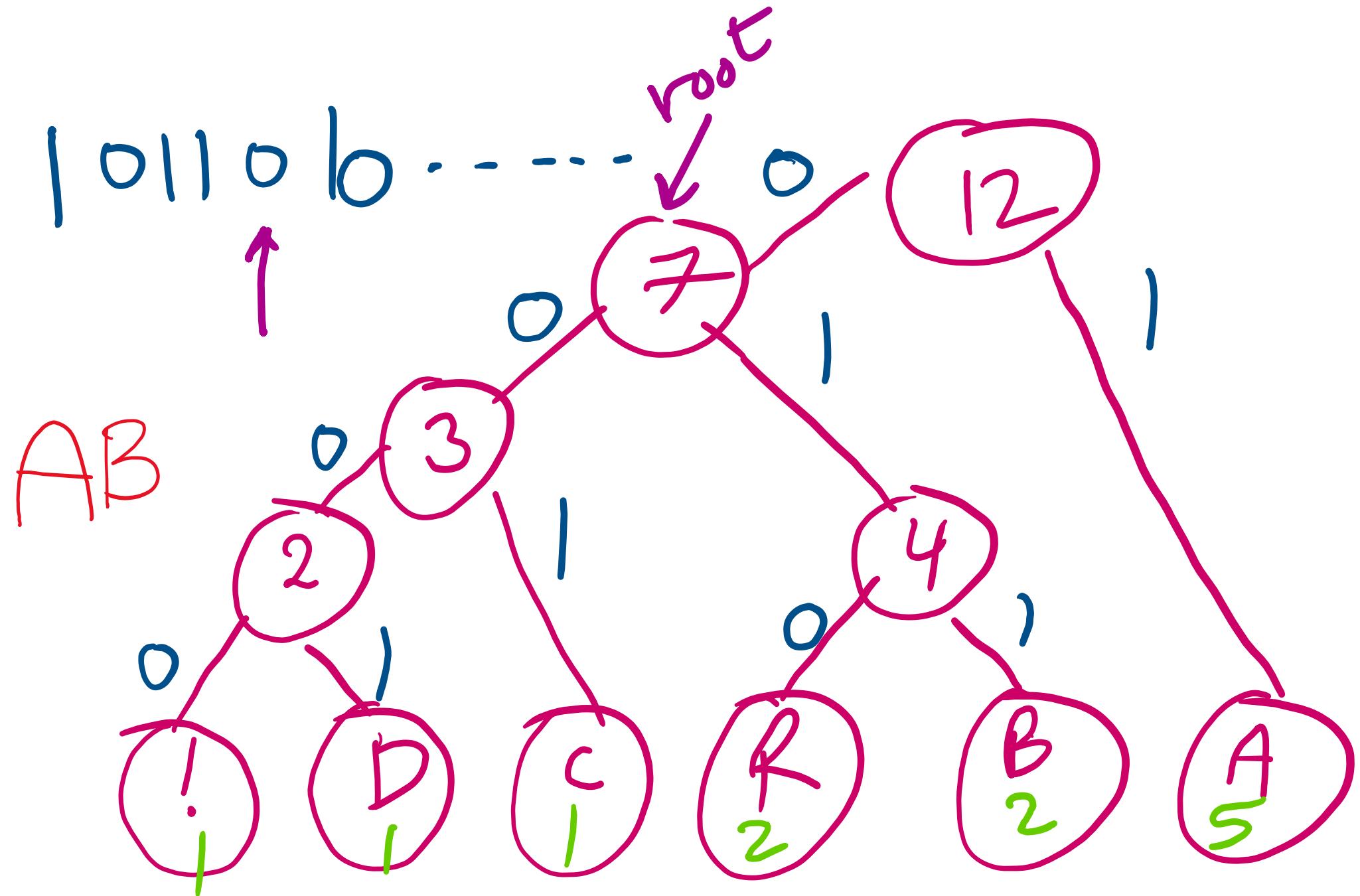
# Huffman Compression: Decoding



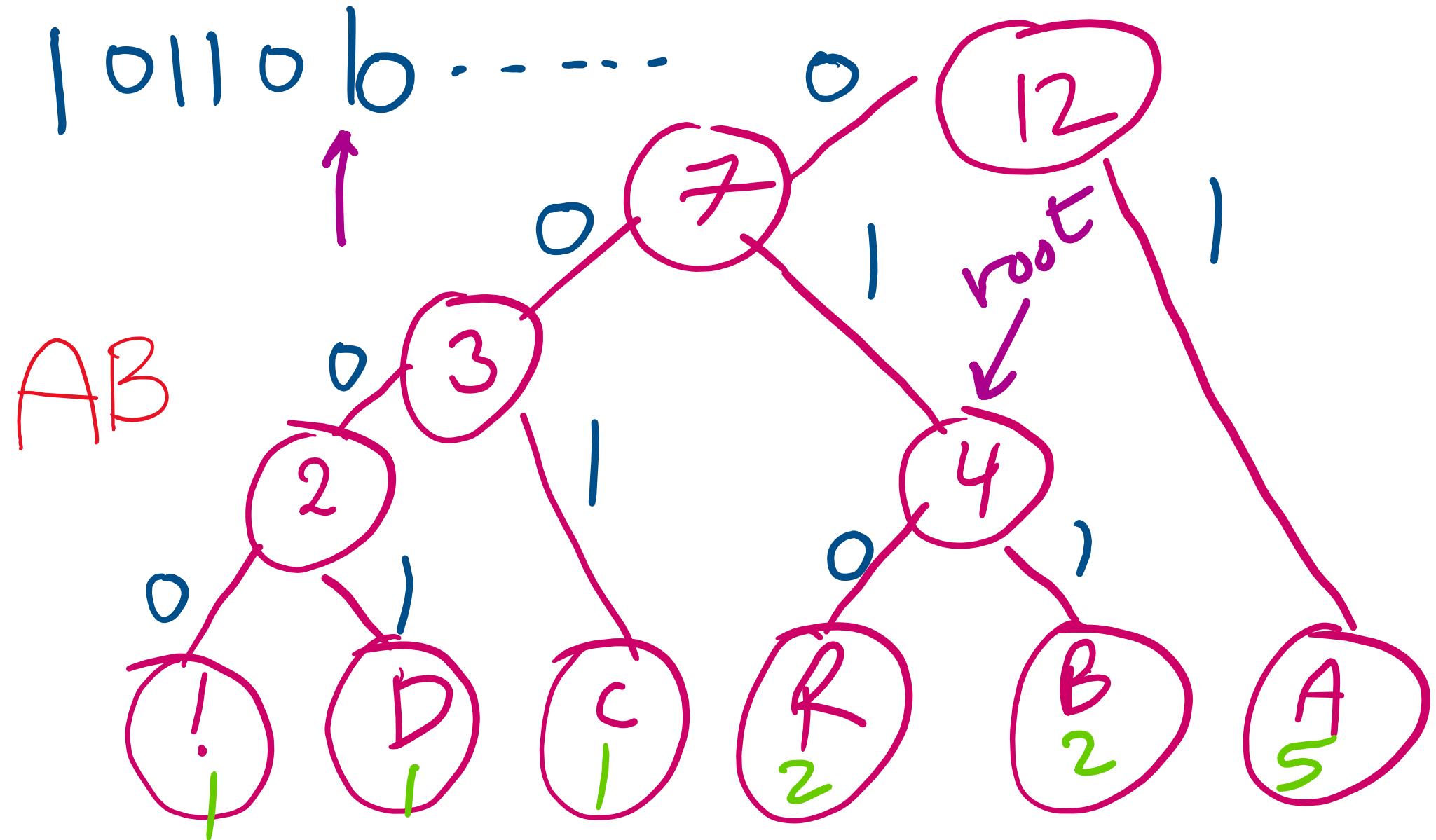
# Huffman Compression: Decoding



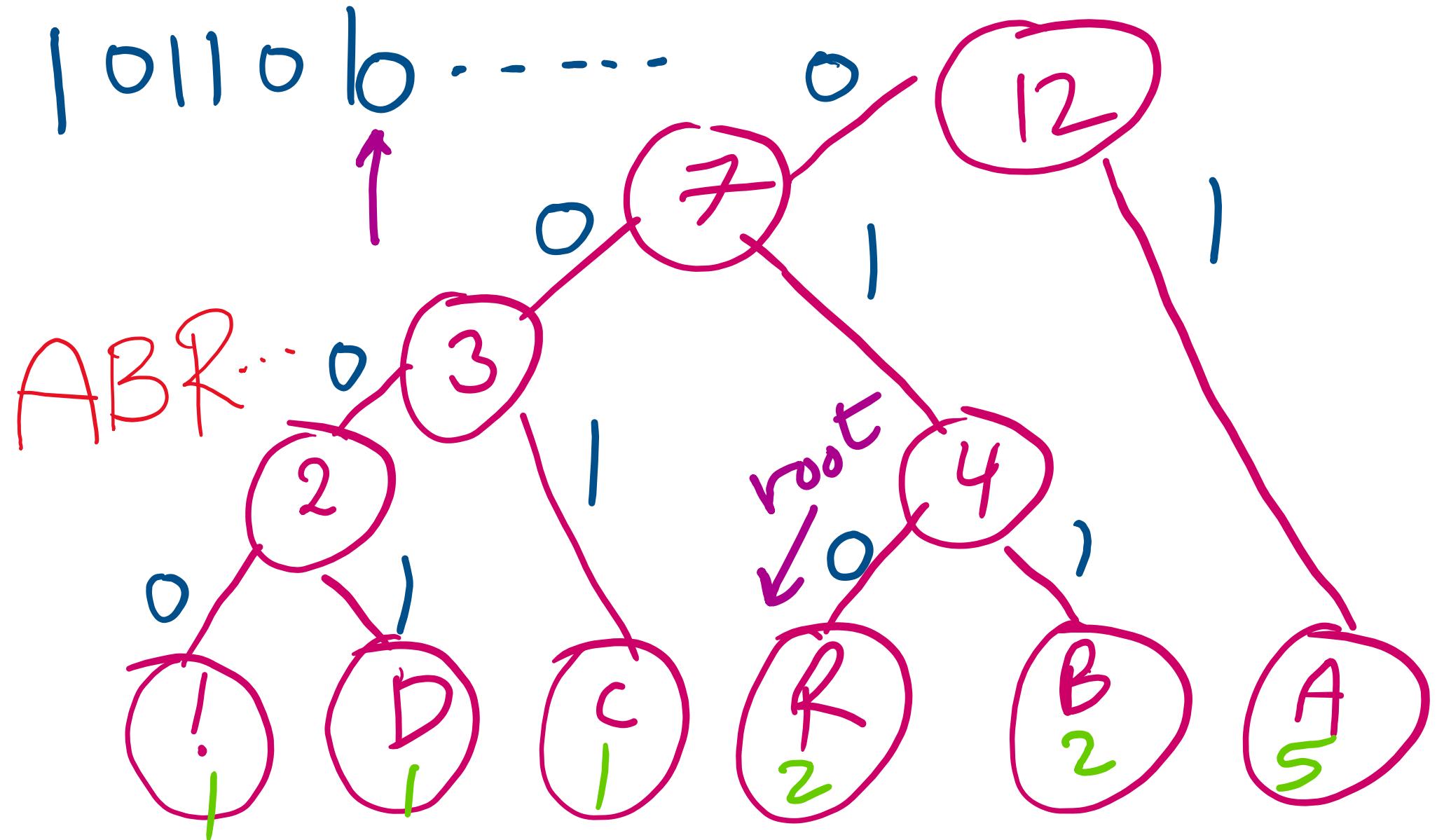
# Huffman Compression: Decoding



# Huffman Compression: Decoding



# Huffman Compression: Decoding



# How do we determine character frequencies?

- Option 1: Preprocess the file to be compressed
  - Upside: Ensure that Huffman's algorithm will produce the best output for the given file
  - Downsides:
    - Requires two passes over the input, one to analyze frequencies/build the trie/build the code lookup table, and another to compress the file
    - Trie must be stored with the compressed file, reducing the quality of the compression
      - This especially hurts small files
      - Generally, large files are more amenable to Huffman compression
        - Just because a file is large, however, does not mean that it will compress well!

# How do we determine character frequencies?

- Option 2: Use a static trie
  - Analyze multiple sample files, build a single tree that will be used for all compressions/expansions
  - Saves on trie storage overhead...
  - But in general not a very good approach
    - Different character frequency characteristics of different files means that a code set/trie that works well for one file could work very poorly for another
      - Could even cause an increase in file size after “compression”!

# How do we determine character frequencies?

- Option 3: Adaptive Huffman coding
  - Single pass over the data to construct the codes and compress a file with no background knowledge of the source distribution
  - Not going to really focus on adaptive Huffman in the class, just pointing out that it exists...

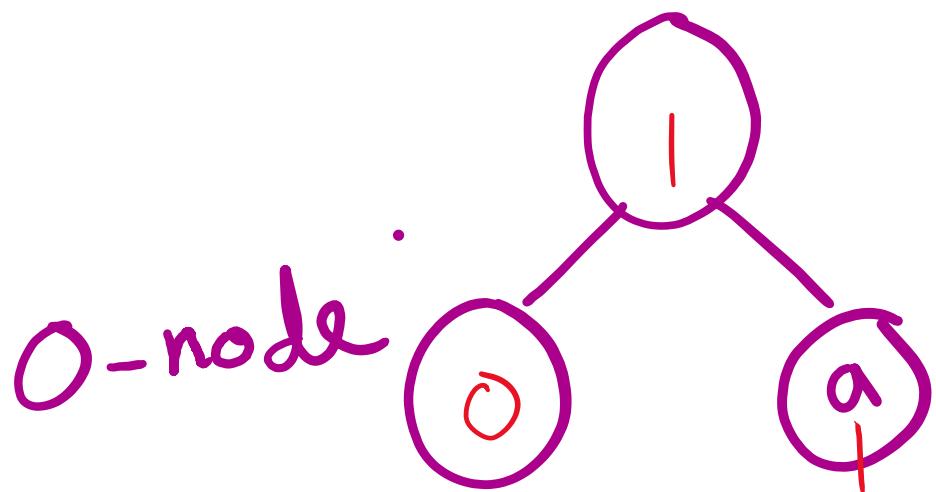
# Adaptive Huffman

a a b b c - - - - - - -



# Adaptive Huffman

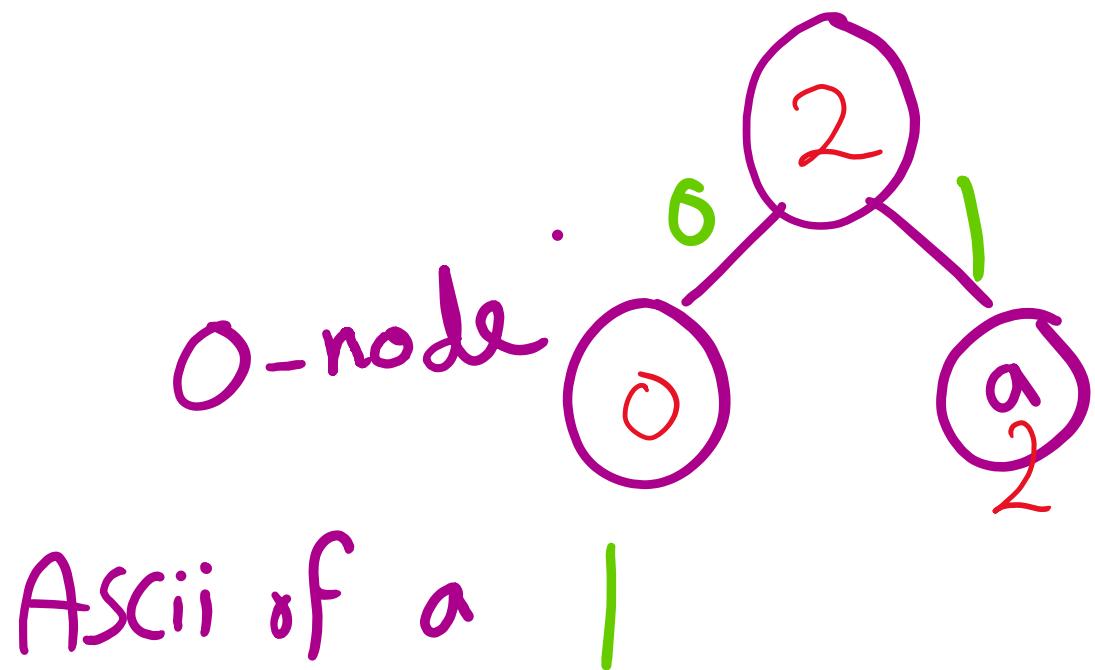
a a b b c - - - - -



Ascii of a

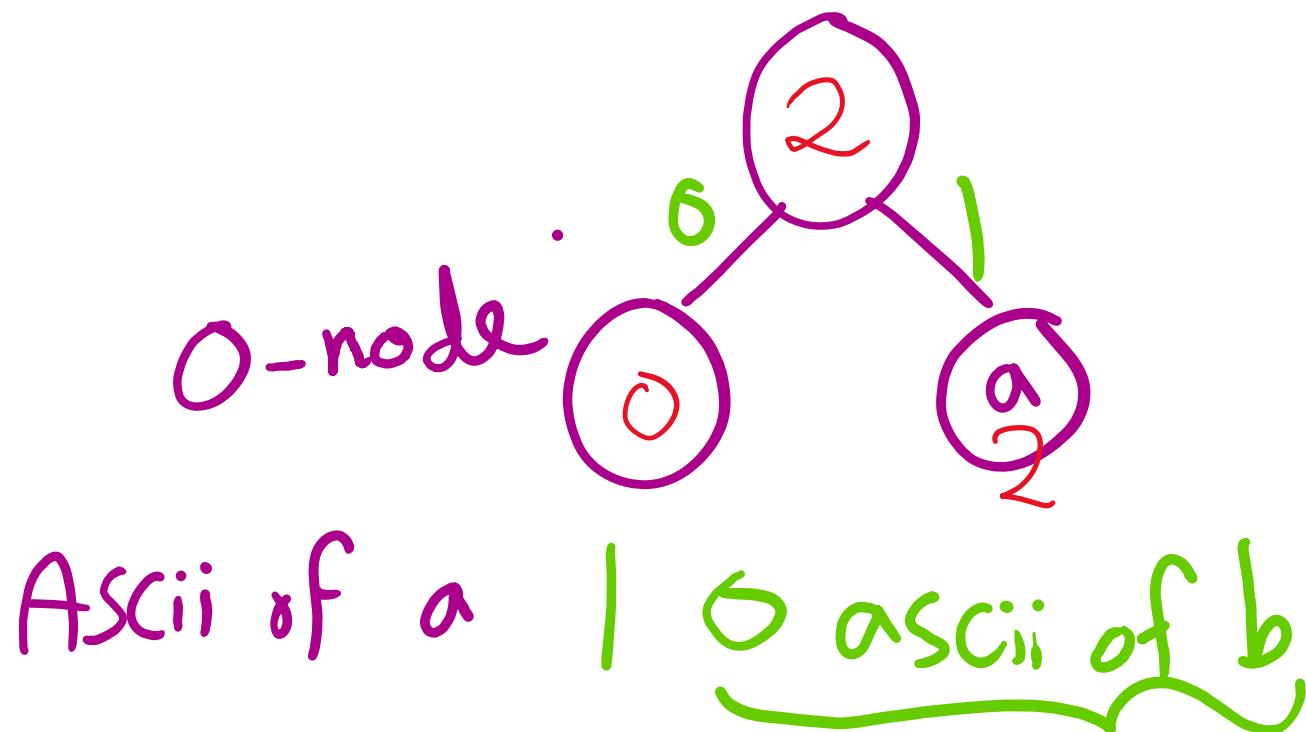
# Adaptive Huffman

a ↴ a b b c - - - - -

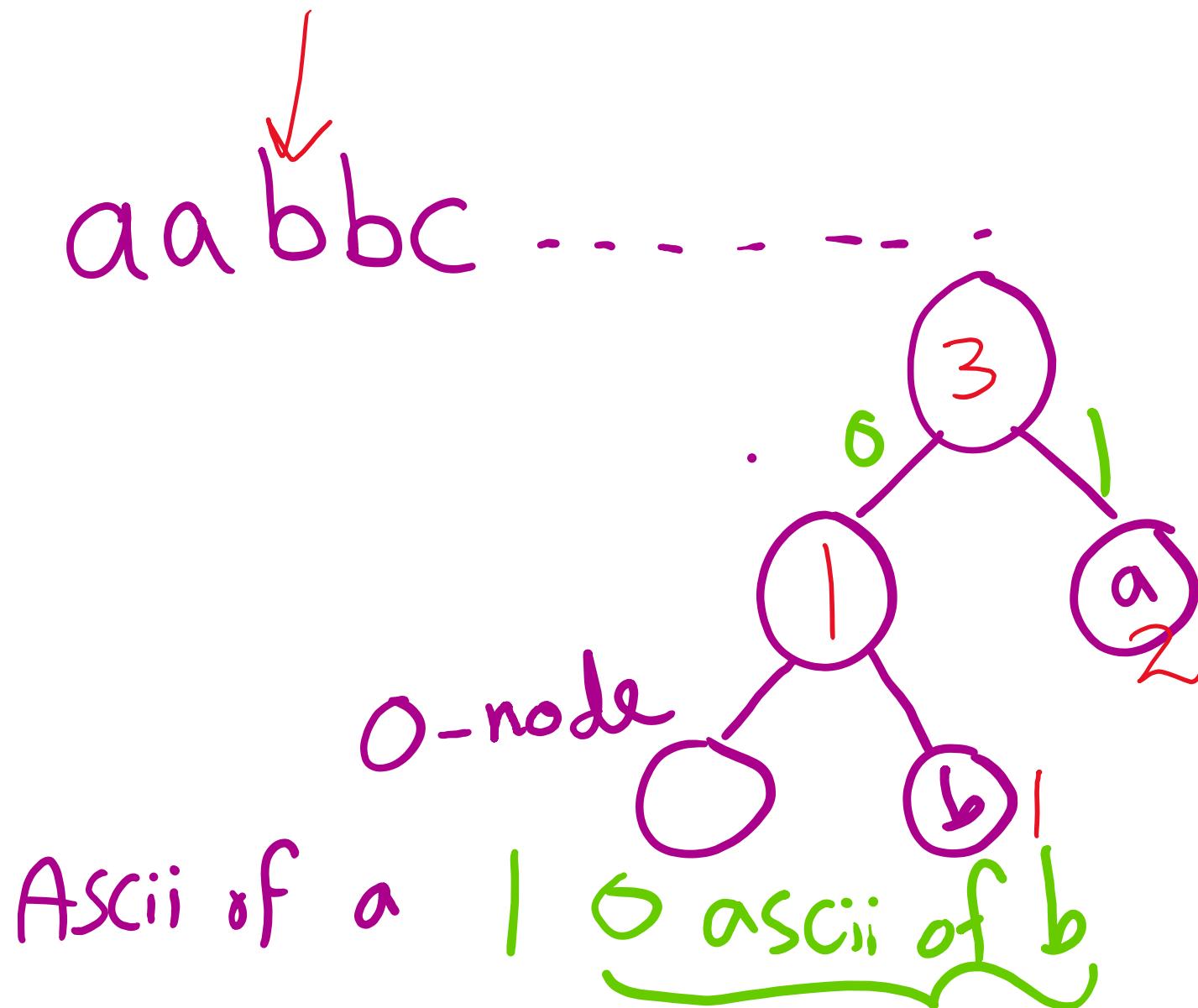


# Adaptive Huffman

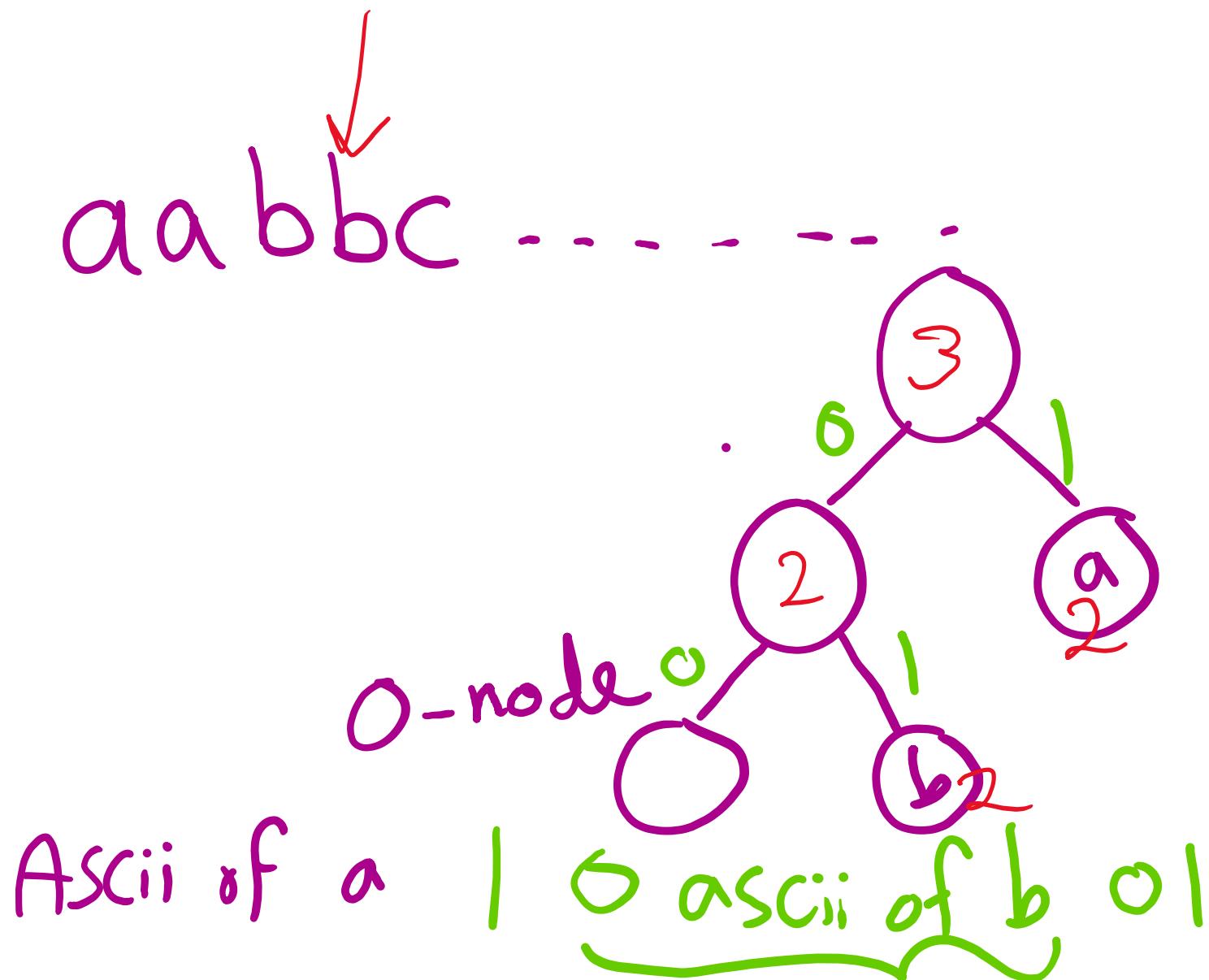
a a b b c - - - - -



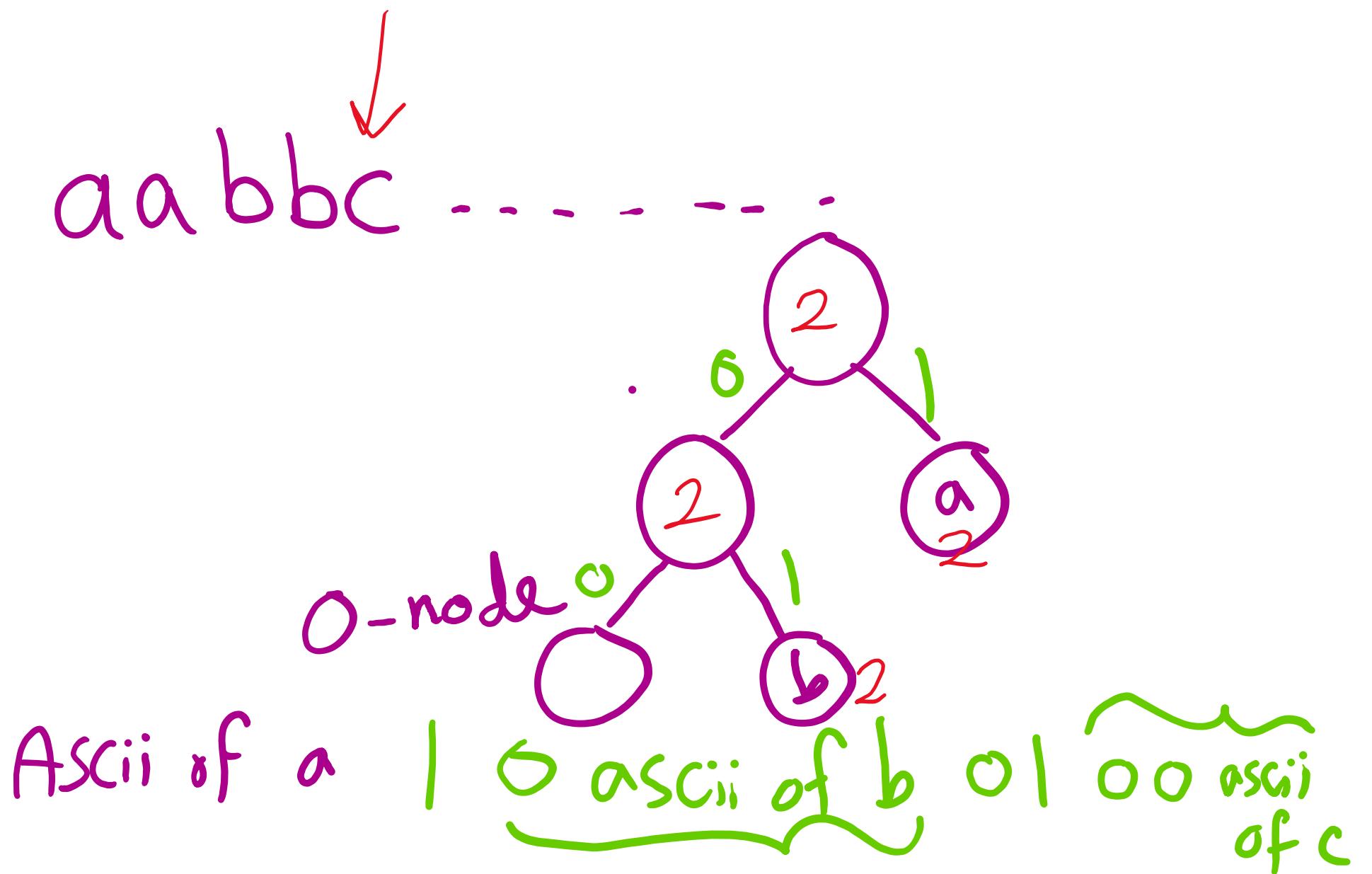
# Adaptive Huffman



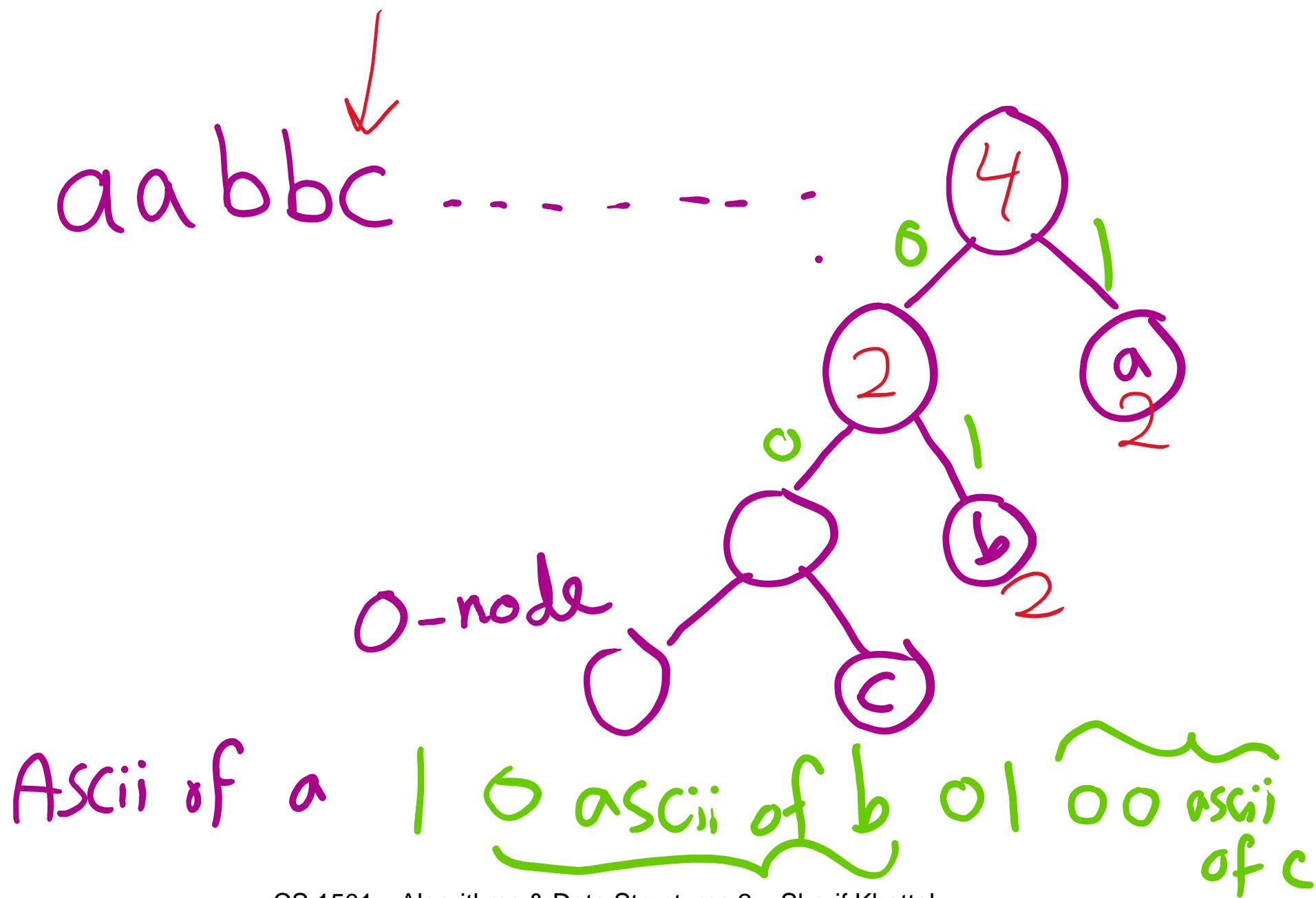
# Adaptive Huffman



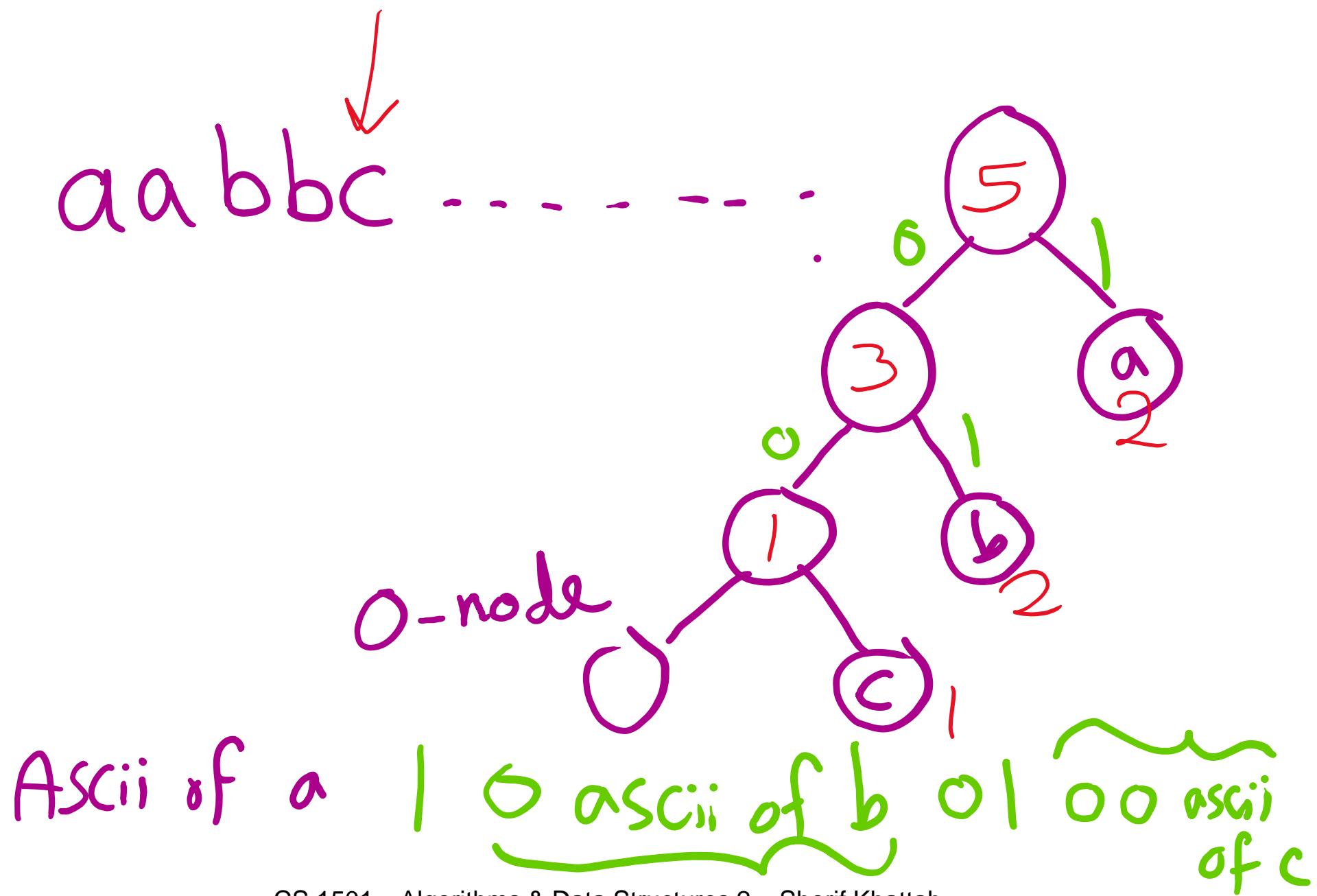
# Adaptive Huffman



# Adaptive Huffman

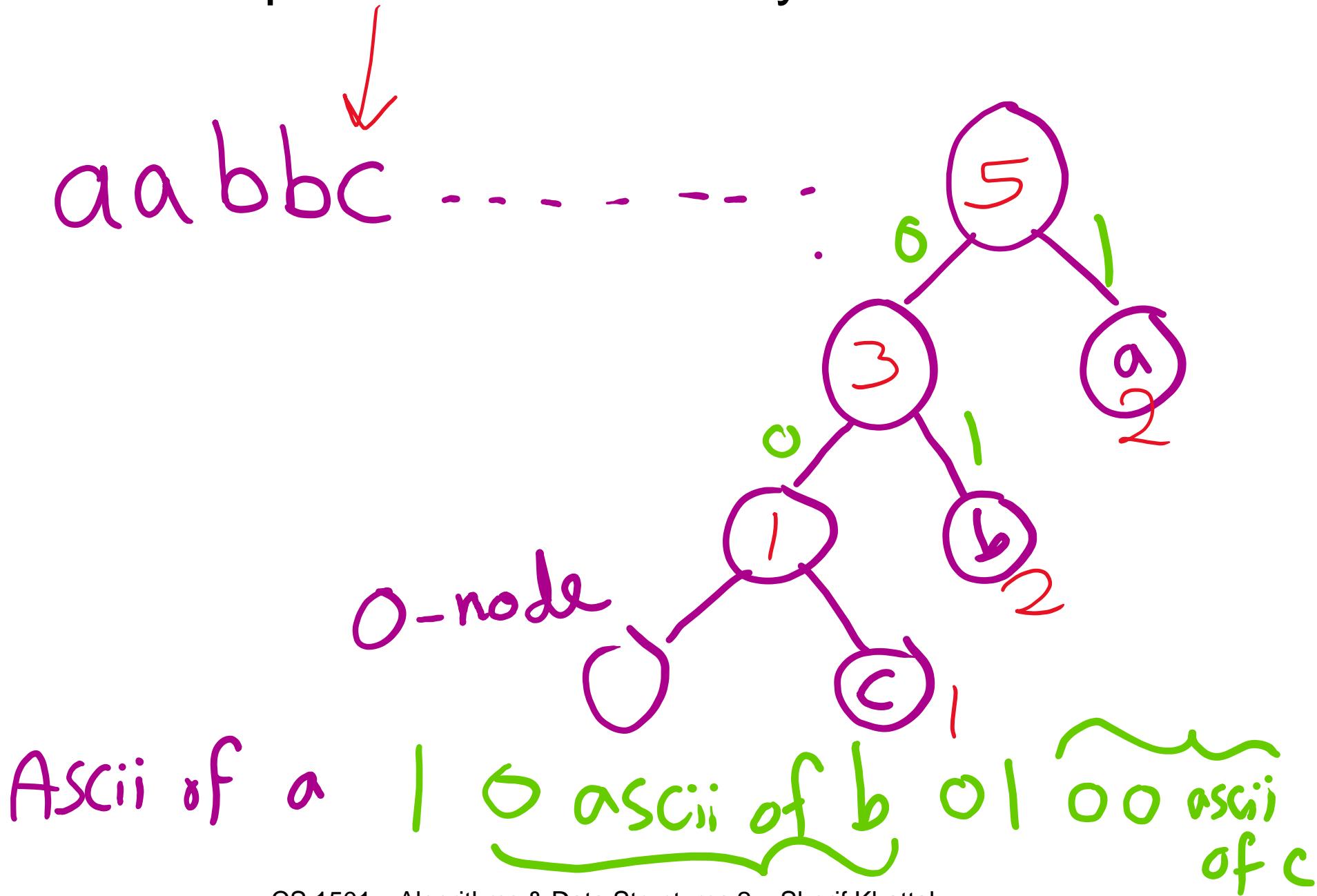


# Adaptive Huffman



# Adaptive Huffman

- need to swap nodes if necessary



# Ok, so how good is Huffman compression

- ASCII requires  $8n$  bits to store  $n$  characters
- For a file containing  $c$  different characters
  - Given Huffman codes  $\{h_0, h_1, h_2, \dots, h_{(c-1)}\}$
  - And frequencies  $\{f_0, f_1, f_2, \dots, f_{(c-1)}\}$
  - Sum from 0 to  $c-1$ :  $|h_i| * f_i$
- Total storage depends on the differences in frequencies
  - The bigger the differences, the better the potential for compression
- Huffman is optimal for character-by-character prefix-free encodings
  - Proof in Propositions T and U of Section 5.5 of the text
- If all characters in the alphabet have the same usage frequency, we can't beat block code (fixed-size codewords)
  - Huffman reduces to fixed-size codewords
  - On a character by character basis...

# That seems like a bit of a caveat...

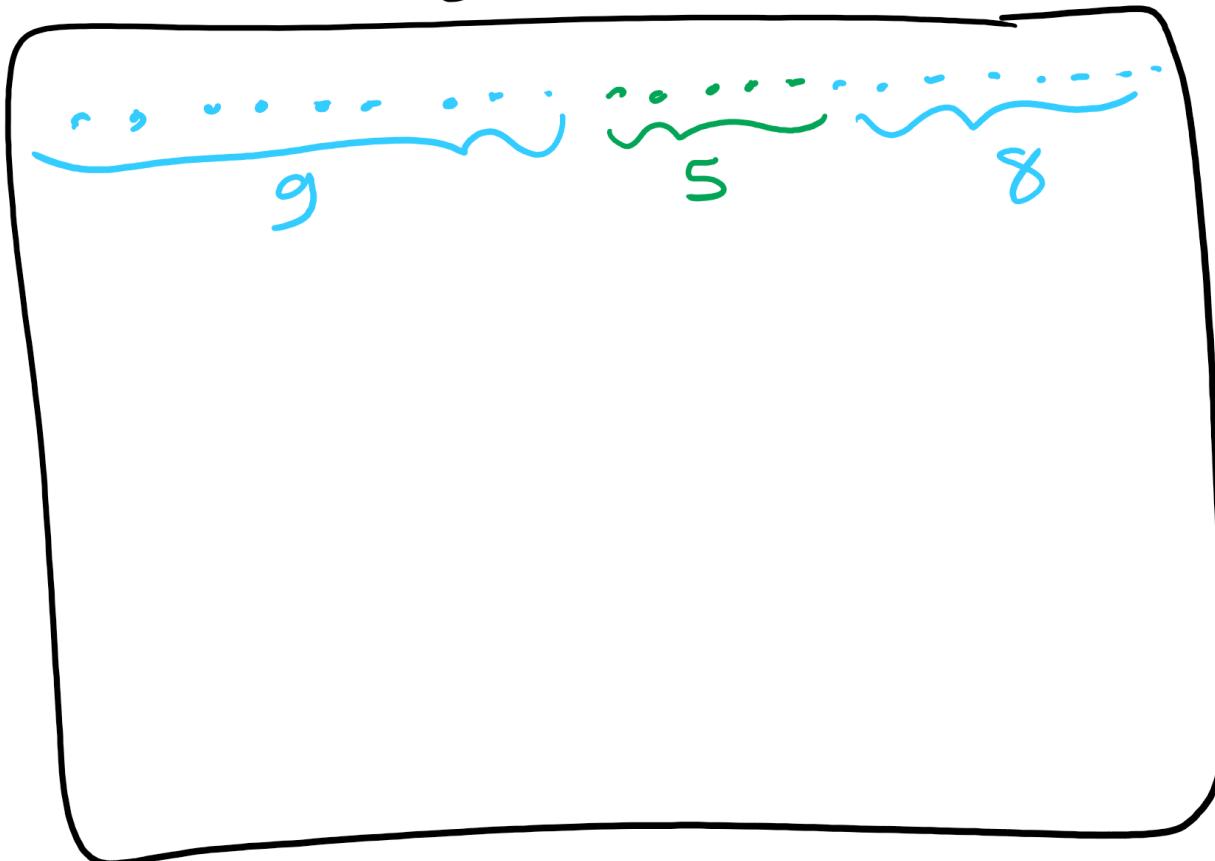
- Where does Huffman fall short?
  - What about repeated patterns of multiple characters?
    - Consider a file containing:
      - 1000 A's
      - 1000 B's
      - ...
      - 1000 of every ASCII character
    - Will this compress at all with Huffman encoding?
      - Nope!
    - But it seems like it should be compressible...

# Run length encoding

- Could represent the previously mentioned string as:
  - 1000A1000B1000C, etc.
    - Assuming we use 10 bits to represent the number of repeats, and 8 bits to represent the character...
      - 4608 bits needed to store run length encoded file
      - vs. 2048000 bits for input file
      - Huge savings!
- Note that this incredible compression performance is based on a very specific scenario...
  - Run length encoding is not generally effective for most files, as they often lack long runs of repeated characters

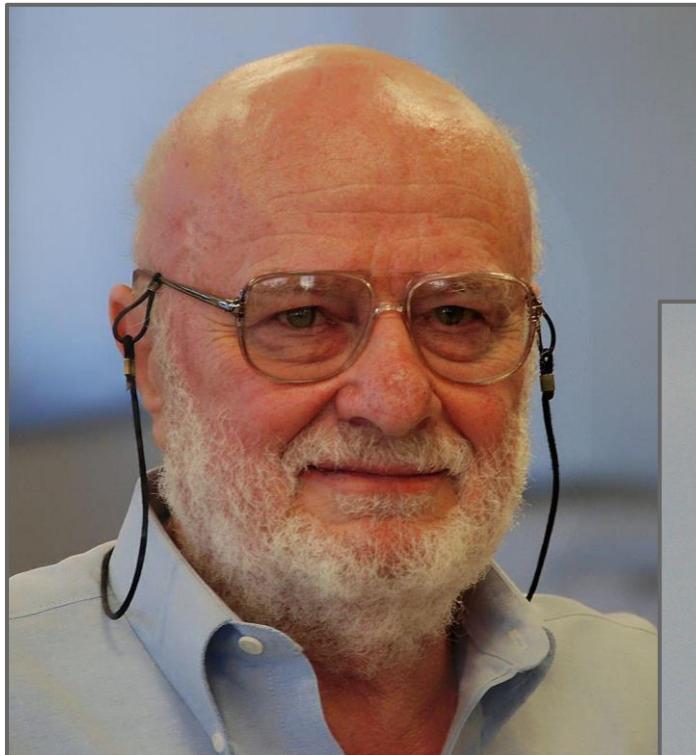
# Run-length Encoding

BW Image



9, 5, 8

# What else can we do to compress files?



# Patterns are compressible, need a general approach

- Huffman used variable-length codewords to represent fixed-length portions of the input...
  - Let's try another approach that uses fixed-length codewords to represent variable-length portions of the input
- Idea: the more characters can be represented by a single codeword, the better the compression
  - Consider "the": 24 bits in ASCII
  - Representing "the" with a single 12 bit codeword cuts the used space in half
    - Similarly, representing longer strings with a 12 bit codeword would mean even better savings!

# How do we know that “the” will be in our file?

- Need to avoid the same problems as the use of a static trie for Huffman encoding...
- So use an adaptive algorithm and build up our patterns and codewords **as we go** through the file

# LZW compression

- Initialize codebook to all single characters
  - e.g., character maps to its ASCII value
  - codewords 0 to 255 are filled now
- While !EOF: (EOF is End Of File)
  - Find the longest match in codebook
  - Output codeword of that match
  - Take this longest match + the next character in the file
    - add that to the codebook with the next available codeword value
  - Start from the character right after the match

# LZW compression example

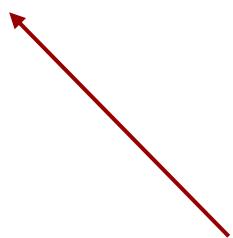
- Compress, using 12 bit codewords:
  - TOBEORNOTTOBEORTOBEORNOT

Cur	Output	Add
T	84	TO:256
O	79	OB:257
B	66	BE:258
E	69	EO:259
O	79	OR:260
R	82	RN:261
N	78	NO:262
O	79	OT:263

T	84	TT:264
TO	256	TOB:265
BE	258	BEO:266
OR	260	ORT:267
TOB	265	TOBE:268
EO	259	EOR:269
RN	261	RNO:270
OT	263	--

# LZW expansion

- Initialize codebook to all single characters
  - e.g., ASCII value maps to its character
- While !EOF:
  - Read next codeword from file
  - Lookup corresponding pattern in the codebook
  - Output that pattern
  - Add the previous pattern + the first character of the current pattern to the codebook



Note this means no codebook addition after first pattern output!

# LZW expansion example

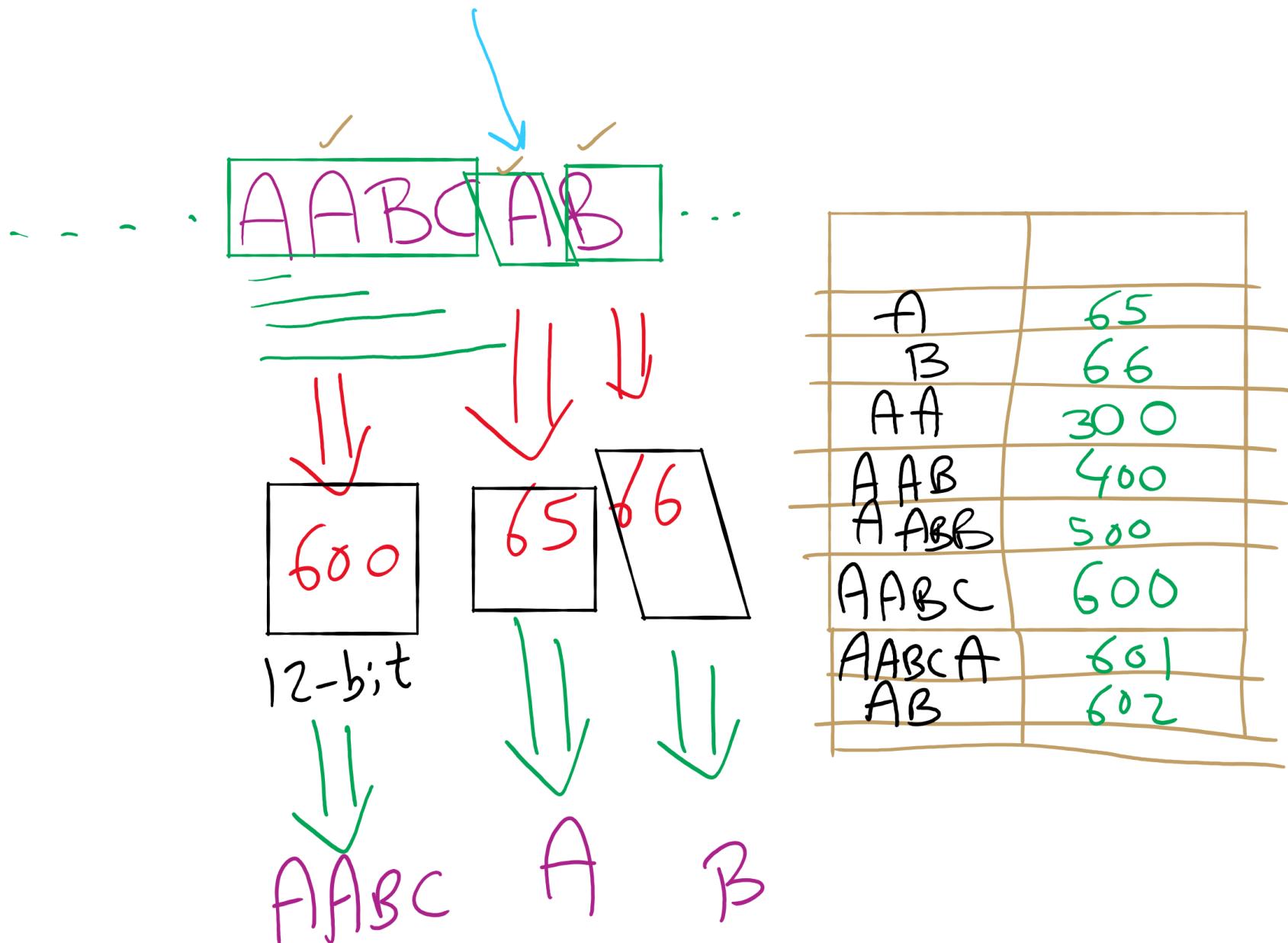
Cur	Output	Add
84	T	--
79	O	256:TO
66	B	257:OB
69	E	258:BE
79	O	259:EO
82	R	260:OR
78	N	261:RN
79	O	262:NO

84	T	263:OT
256	TO	264:TT
258	BE	265:TOB
260	OR	266:BEO
265	TOB	267:ORT
259	EO	268:TOBE
261	RN	269:EOR
263	OT	270:RNO

# How does this work out?

- Both compression and expansion construct the same codebook!
  - Compression stores character string → codeword
  - Expansion stores codeword → character string
  - They contain the same pairs in the same order
    - Hence, the codebook doesn't need to be stored with the compressed file, saving space

# LZW Example



# Just one tiny little issue to sort out...

- Expansion can sometimes be a step ahead of compression...
  - If, during compression, the (pattern, codeword) that was just added to the dictionary is immediately used in the next step, the decompression algorithm will not yet know the codeword.
  - This is easily detected and dealt with, however

# LZW corner case example

- Compress, using 12 bit codewords: AAAAAAA

Cur	Output	Add
A	65	AA:256
AA	256	AAA:257
AAA	257	--

- Expansion:

Cur	Output	Add
65	A	--
256	AA	256:AA
257	AAA	257:AAA

# LZW Corner Case

A A AB BB

↓ ↓ ↓ ↓

65 65 66 258

↓ ↓ ↓ ↓

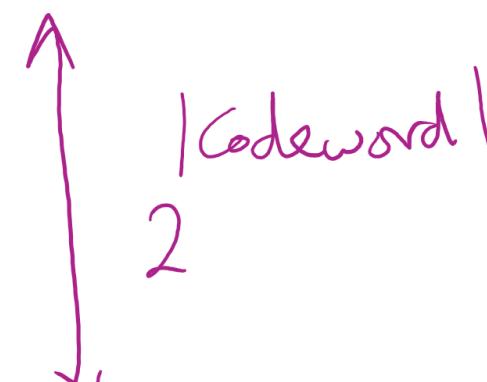
J J J J

A A B BB

A	65
AA	256
AB	257
BB	258

65	A
66	B
256	AA
257	AB
258	BB

Prev output + 1<sup>st</sup> character of  
} Prev output }



# LZW implementation concerns: codebook

- How to represent/store during:
  - Compression
  - Expansion
- Considerations:
  - What operations are needed?
  - How many of these operations are going to be performed?
- Discuss

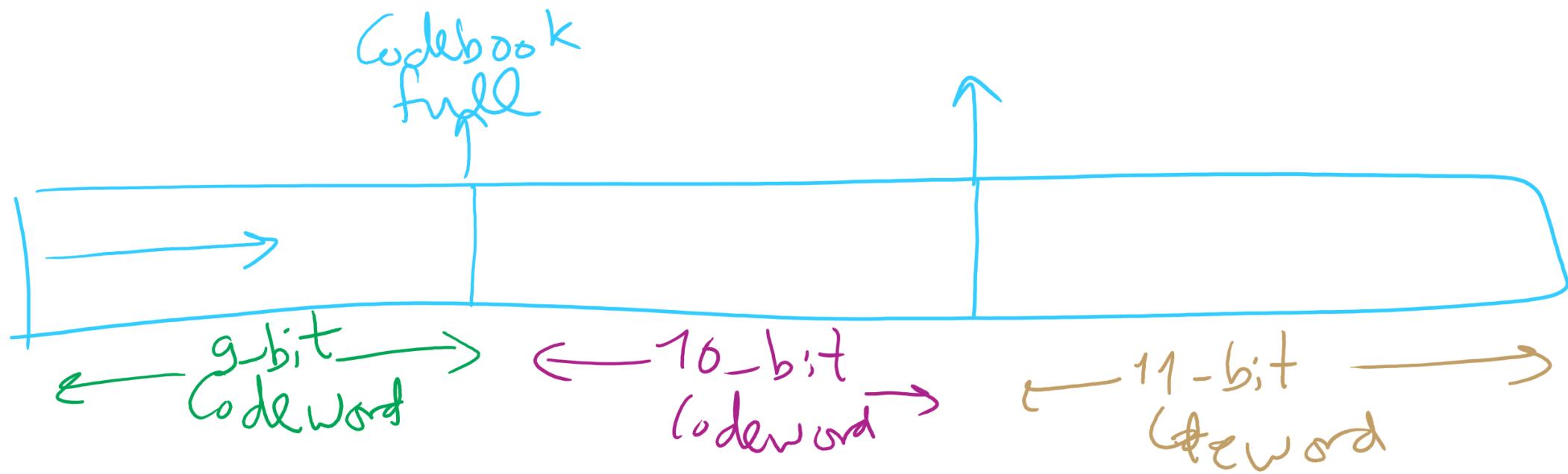
# Further implementation issues: codeword size

- How long should codewords be?
  - Use fewer bits:
    - Gives better compression earlier on
    - But, leaves fewer codewords available, which will hamper compression later on
  - Use more bits:
    - Delays actual compression until longer patterns are found due to large codeword size
    - More codewords available means that greater compression gains can be made later on in the process

# Variable width codewords

- This sounds eerily like variable length codewords...
  - Exactly what we set out to avoid!
- Here, we're talking about a different technique
- Example:
  - Start out using 9 bit codewords
  - When codeword 512 is inserted into the codebook, switch to outputting/grabbing 10 bit codewords
  - When codeword 1024 is inserted into the codebook, switch to outputting/grabbing 11 bit codewords...
  - Etc.

# Adaptive Codeword Size

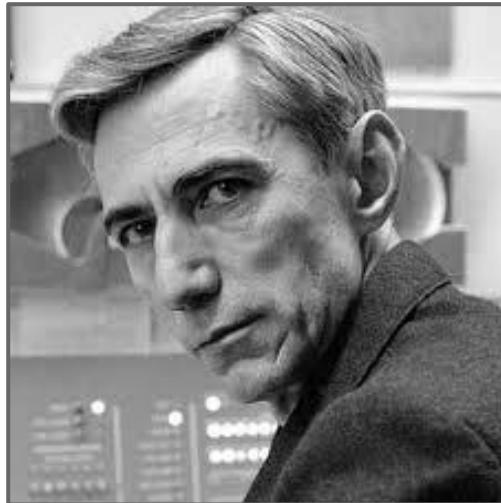


# Even further implementation issues: codebook size

- What happens when we run out of codewords?
  - Only  $2^n$  possible codewords for  $n$  bit codes
  - Even using variable width codewords, they can't grow arbitrarily large...
- Two primary options:
  - Stop adding new keywords, use the codebook as it stands
    - Maintains long already established patterns
    - But if the file changes, it will not be compressed as effectively
  - Throw out the codebook and start over from single characters
    - Allows new patterns to be compressed
    - Until new patterns are built up, though, compression will be minimal

# Can we reason about how much a file can be compressed?

- Yes! Using Shannon Entropy



# Information theory in a single slide...

- Founded by Claude Shannon in his paper "A Mathematical Theory of Communication"
- *Entropy* is a key measure in information theory
  - Slightly different from thermodynamic entropy
  - A measure of the unpredictability of information content
  - Example: which is more unpredictable?
    - a character that occurs with a probability of 0.5 or
    - a character that occurs with probability 0.25
    - which should have more entropy?

# Entropy

- Entropy equation:  $H(c) = -1 * \log_2 \Pr(c)$ 
  - $\Pr(c)$  is the probability of character  $c$
- Examples:
  - $\Pr(c1) = 0.5 \rightarrow H(c1) = -1 * \log_2(0.5) = -1*-1 = 1$  bit
  - $\Pr(c2) = 0.25 \rightarrow H(c2) = -1*\log_2(0.25) = -1*-2 = 2$  bits
  - $\Pr(c3) = 1/2^{100} \rightarrow H(c3) = -1*\log_2(2^{-100}) = -1*-100 = 100$  bits

# Implications on Lossless Compression

- On average, a lossless compression scheme cannot compress a message to have more than 1 bit of entropy per bit of compressed message
- By losslessly compressing data, we represent the same information in less space
  - entropy of 8 bits of compressed data > entropy of 8 bits of uncompressed data

# Entropy of a file

- The average number of bits required to store a character in that file
- So, it is the average entropy of all unique characters in the file
- $H(\text{file}) = \sum_{\text{each unique character } c} H(c) * Pr(c)$
- How can we determine the probability of each character in the file?
  - if depends only on file contents
    - $Pr(c) = f(c) / \text{file size}$
  - However, may also depend on receiver and sender contexts and their world knowledge

# Entropy applied to language:

- the average number of bits required to store a letter of the language
- Entropy of a language \* length of message = amount of information contained in that message
- Uncompressed, English has between 0.6 and 1.3 bits of entropy per letter

# The showdown you've all been waiting for...

## HUFFMAN vs LZW

- In general, LZW will give better compression
  - Also better for compressing archived directories of files
    - Why?
      - Very long patterns can be built up, leading to better compression
      - Different files don't "hurt" each other as they did in Huffman
        - Remember our thoughts on using static tries?

# So lossless compression apps use LZW?

- Well, gifs can use it
  - And pdfs
- Most dedicated compression applications use other algorithms:
  - DEFLATE (combination of LZ77 and Huffman)
    - Used by PKZIP and gzip
  - Burrows-Wheeler transforms
    - Used by bzip2
  - LZMA
    - Used by 7-zip
  - brotli
    - Introduced by Google in Sept. 2015
    - Based around a " ... combination of a modern variant of the LZ77 algorithm, Huffman coding[,] and 2nd order context modeling ..."

# Is there a universal compression algorithm?

- Nope!
- No algorithm can compress every bitstream
  - Assume we have such an algorithm
  - We can use to compress its own output!
  - And we could keep compressing its output until our compressed file is 0 bits!
    - Clearly this can't work
- Proofs in Proposition S of Section 5.5 of the text

# Is finding the best algorithm for a given file possible?

- Nope!
- This problem is undecidable
- Example:
  - A Fibonacci sequence of one billion numbers can be compressed by a program to generate Fibonacci numbers

# A final note on compression evaluation

- "Weissman scores" are a made-up metric for Silicon Valley (TV)

